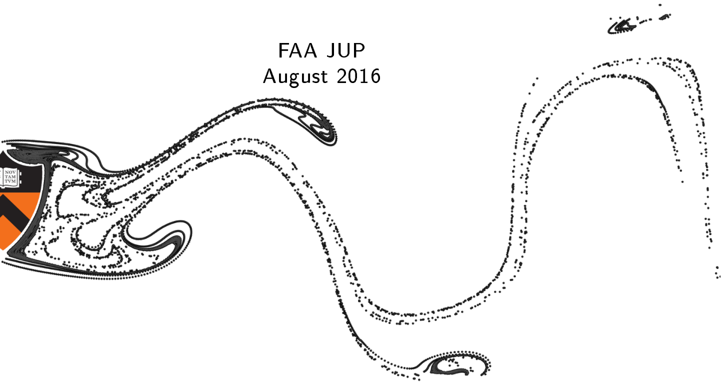
Continuing progress of high-order accurate simulation tool for cargo hold fires

Mark Lohry
Princeton University

FAA JUP
August 2016



Outline

Motivation

- FAA requirement for alarms to go off within 60 seconds of fire ignition.
- Several different detection methods are generally used together, e.g. temperature, smoke/particulate, radiation, optical
- Their effectiveness is determined by the dynamics of a particular fire and their relative position.
- Accurate prediction of fire-induced flow in a cargo hold is a necessary first step to predicting detection capabilities.
- More reliable detection capabilities could potentially reduce false alarms.

B707 cargo geometry

- Experimental and computational data for B707 cargo fires available from work at Sandia and FAA Tech center.
- Current goal is to perform a direct comparison of those results with our new solver.

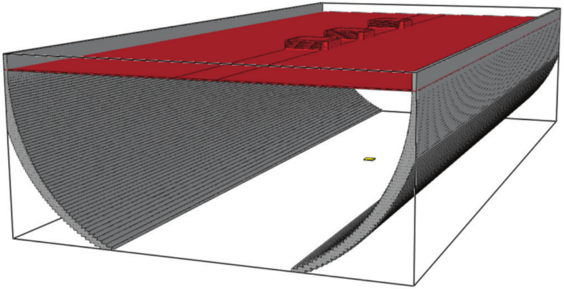


Figure : B707 cargo hold geometry.

Fire-induced fluid dynamics

- Detailed simulation of the combustion process is expensive and unnecessary; the large scale dynamics are primarily determined by the amount of heat release, its position, and the geometry.
- Commonly used models apply a heat source and input of reaction products (CO, CO2, etc.)

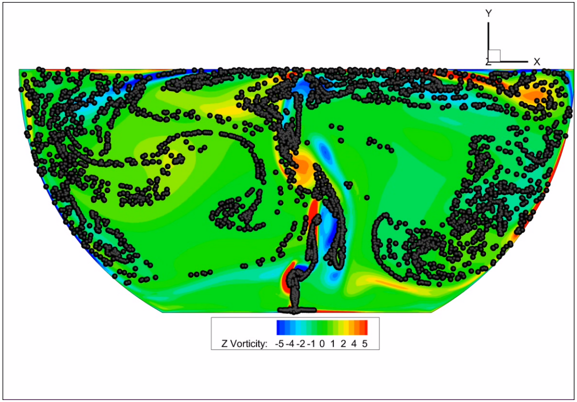


Figure : Flow driven by an enclosed heat source.

Cluttered geometry 2D

- A real fire is unlikely to happen in an empty cargo hold.
- Including some obstructions changes the flowfield considerably.

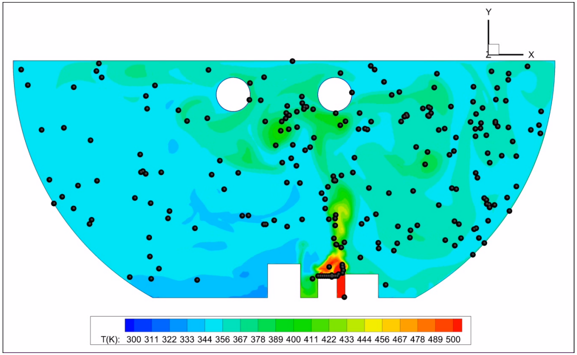


Figure : $t = 20s$ after ignition.

Simulation challenges

Simulating a single fire case is relatively straightforward, but of limited utility. There are several uncertainties to address:

- Initial position, size, and strength of a fire is unknown.
- Cargo hold geometry varies considerably depending on contents.

Simulation needs:

- Complex geometries: must handle complex boundary conditions accurately.
- Fast: uncertainty quantification will require a large number of simulations.
- Accurate: must accurately simulate vorticity-dominated turbulent flows for transport prediction.

Available tools

FDS: NIST's Fire Dynamics Simulator.

- **Pros:**
  - Purpose-built for smoke and heat transport from fires using large eddy simulation.
  - Combustion and radiation models.
  - Built-in post-processing tools related to smoke transport.
- **Cons:**
  - Handles complex boundaries with Cartesian cut cells: inaccurate for anything but rectangles.

OpenFOAM

- **Pros:**
  - Similar combustion and radiation models to FDS, with additional thermodynamic models.
  - Handles arbitrary body-fitted meshes.
  - Wide array of LES models.
- **Cons:**
  - Very slow for large cases.

Fluent

- **Pros:**
  - Well known, full combustion and radiation modeling.
  - Handles arbitrary body-fitted meshes.
  - Wide array of LES models.
- **Cons:**
  - Commercial

All limited to $O(\Delta x^2)$ accuracy.

High order accurate CFD

- Even very low intensity fires will have very complex flow phenomena poorly captured by low-order CFD methods.

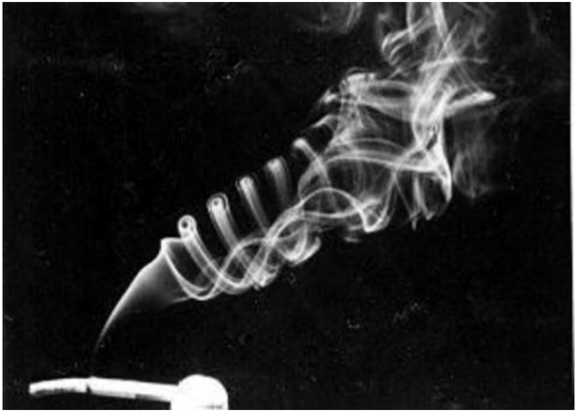


Figure : Instability of smoke from a cigarette, *Perry & Lim, 1978*

High order accurate CFD

Order of accuracy in finite differences:

$$\begin{aligned}
\frac{du}{dx} &\approx \frac{u(x+\Delta x) - u(x)}{\Delta x} + O(\Delta x) \\
\frac{du}{dx} &\approx \frac{u(x) - u(x-\Delta x)}{\Delta x} + O(\Delta x) \\
\frac{du}{dx} &\approx \frac{u(x+\Delta x) - u(x-\Delta x)}{2\Delta x} + O(\Delta x^2)
\end{aligned} \tag{1}$$

- Error scales like $\sim O(\Delta x^n)$ for order n.
- For a 1^{st} order method, halving the grid spacing reduces error by $\sim 1/2$.
- For a 4^{th} order method, halving the grid spacing reduces error by $\sim 1/16$.

High order accurate CFD

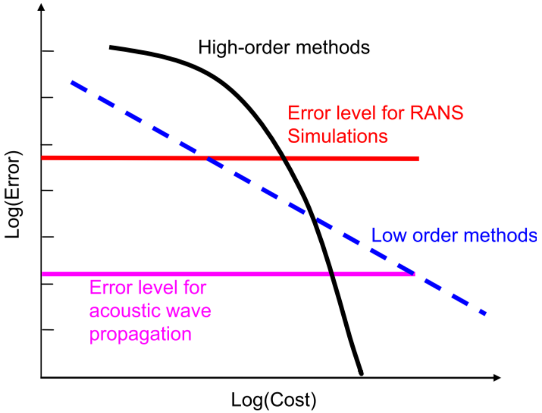


Figure : Generic error vs cost plot, *Wang, 2007*

Discontinuous Galerkin discretization method

For a multi-dimensional conservation law

$$\frac{\partial u(\mathbf{x},t)}{\partial t} + \nabla \cdot \mathbf{f}(u(\mathbf{x},t),\mathbf{x},t) = 0 \tag{2}$$

approximate $u(\mathbf{x},t)$ by

$$u(\mathbf{x},t) \approx u_h(\mathbf{x},t) = \sum_{i=1}^{N_p} u_h(\mathbf{x}_i,t) l_i(\mathbf{x}) = \sum_{i=1}^{N_p} \hat{u}_i(t)\psi_i(\mathbf{x}) \tag{3}$$

where $l_i(\mathbf{x})$ is the multidimensional Lagrange interpolating polynomial defined by grid points $\mathbf{x}_i$, N_p is the number of nodes in the element, and $\psi_i(\mathbf{x})$ is a local polynomial basis.

- Of the two equivalent approximations here, the first is termed *nodal* and the second *modal*. i.e., u_h represents values of u at discrete nodes with a reconstruction based on Lagrange polynomials, and $\hat{u}_i$ represents modes/coefficients for reconstruction with the basis ψ_n.

Discontinuous Galerkin discretization method

Substituting the approximation u_h into the conservation law:

$$\frac{\partial u_h}{\partial t} + \nabla \cdot \mathbf{f}_h = 0$$

Integrate with a test function ψ_j, the same as used to represent the polynomial above,

$$\int_V \frac{\partial u_h}{\partial t}\psi_j \, dV + \int_V \nabla \cdot \mathbf{f}_h \psi_j \, dV = 0$$

Integration by parts on the spatial component:

$$\int_V \frac{\partial u_h}{\partial t}\psi_j \, dV - \int_V \nabla\psi_j \cdot \mathbf{f}_h \, dV + \oint_S \psi_j {\mathbf{f}^\star}_h \cdot \mathbf{n} \, dS = 0$$

Using the modal representation, $u_h = \sum_{i=1}^{N_p} \hat{u}_i(t)\psi_i(\mathbf{x})$

$$\int_V \frac{\partial \hat{u}_i\psi_i}{\partial t}\psi_j \, dV - \int_V \nabla\psi_j \cdot \hat{\mathbf{f}}_i\psi_i \, dV + \oint_S \psi_j \hat{\mathbf{f}}_i^\star \psi_i \cdot \mathbf{n} \, dS = 0$$

which gives the semi-discrete form of the classic modal DG method,

$$\hat{\mathbf{M}}_{ij}\frac{d\,\hat{u}_i}{dt} = \int_V \nabla\psi_j \cdot \hat{\mathbf{f}}_i\psi_i \, dV + \oint_S \psi_j \hat{\mathbf{f}}_i^\star \phi_i \cdot \mathbf{n} \, dS$$

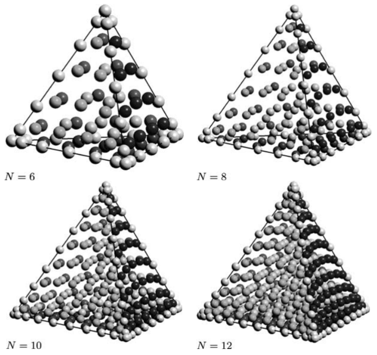
Here $\mathbf{M}$ is the *mass matrix* (identity for orthonormal bases), $\mathbf{n}$ the vector normal at an element surface, and $\hat{\mathbf{f}}^\star$ is a conservative flux function at interfaces, equivalent to that used in finite volume methods.

Discontinuous Galerkin discretization method

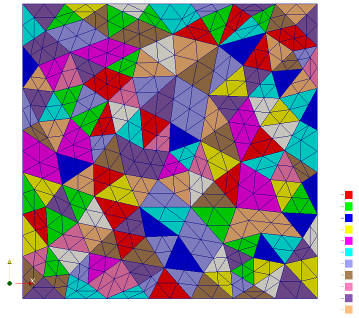
The modal coefficients $\hat{\mathbf{u}}$ can always be represented on nodal locations $\mathbf{u}$ through a change of basis by the *Vandermonde matrix*,

$$V\hat{\mathbf{u}} = \mathbf{u}$$

which turns the previous modal method into a nodal method. This code uses unstructured tetrahedral elements in 3D with Legendre-Gauss-Lobatto nodes:



(a) Volume nodes for varying order, Hesthaven & Warburton.



(b) $N = 2$ element surfaces; nodes are at line intersections.

Discretization method - solving the discretized equations

This ends up with a potentially very large system of ODEs to be solved:

$$\frac{d\mathbf{u}}{dt} = \mathbf{f}(\mathbf{u}, \mathbf{u}', \mathbf{t})$$

Simplest method for integrating this system in time is the explicit (forward) Euler method:

$$\mathbf{u}^{n+1} = \mathbf{u}^{n} + \Delta t \mathbf{f}(\mathbf{u}, \mathbf{u}', \mathbf{t})^{\mathbf{n}}$$

Unfortunately, explicit time-stepping for high-order DG is stable only for excessively small Δt,

$$\Delta t = O(\frac{\Delta x}{N^2})$$

where a mesh cell Δx can be very small (boundary layers, small geometric features) and N^2 quickly grows large. For any engineering-scale problem, explicit methods are unfeasible for use.

- This requires the use of implicit time-stepping methods, e.g. 1st order backward Euler:

$$\mathbf{u}^{n+1} = \mathbf{u}^{n} + \Delta t \mathbf{f}(\mathbf{u}, \mathbf{u}', \mathbf{t})^{\mathbf{n+1}}$$

where we now have a set of non-linear equations to solve for u^{n+1}. **Typically we use 3rd order or higher time-accurate schemes.**

Discretization method - solving the discretized equations

Task is to solve the very large non-linear system at each time step:

$$\mathbf{F}(\mathbf{u}) = 0$$

Newton's method for this problem derives from a Taylor expansion (Knoll/Keyes 2004):

$$\mathbf{F}(\mathbf{u}^{k+1}) = \mathbf{F}(\mathbf{u}^{k}) + \mathbf{F}'(\mathbf{u}^{k})(\mathbf{u}^{k+1} - \mathbf{u}^{k})$$

resulting in a sequence of linear systems

$$\mathbf{J}(\mathbf{u}^{k})\delta\mathbf{u}^{k} = -\mathbf{F}(\mathbf{u}^{k}), \quad \mathbf{u}^{k+1} = \mathbf{u}^{k} + \delta\mathbf{u}^{k}$$

for the Jacobian $\mathbf{J}$.

- The linear system $\mathbf{J}(\mathbf{u}^{k})\delta\mathbf{u}^{k} = -\mathbf{F}(\mathbf{u}^{k})$ is straightforward enough to write, but for these methods $\mathbf{J}$ is a very large sparse matrix which is prohibitively expensive to actually compute and store.
- A mesh of 100,000 4th order cells requires roughly 250GB of memory to store in 64-bit floats.

Discretization method - solving the discretized equations

- A remedy for this is to use a “Jacobian-Free” method based on Krylov subspace iterations (e.g. GMRES, BiCGSTAB), which only require the action of the jacobian in the form of matrix-vector products:

$$\mathbf{K} = \text{span}(\mathbf{J}\delta\mathbf{r}, \mathbf{J}^2\delta\mathbf{r}, \mathbf{J}^3\delta\mathbf{r}, ...)$$

which can be approximated by a finite difference:

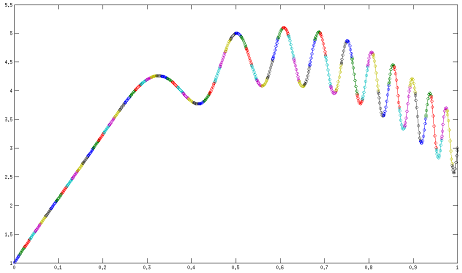
$$\mathbf{Jv} \approx [\mathbf{F}(\mathbf{u} + \epsilon\mathbf{v}) - \mathbf{F}(\mathbf{v})]/\epsilon$$

- This enables a solution method for the non-linear system that doesn’t require ever explicitly forming the Jacobian, and instead only requires the evaluation of the RHS of the ODE.
- This is the *Jacobian-free Newton-Krylov* (JFNK) method:
  - Take a Newton step from the previous iterate.
  - Approximately solve the linear system using a matrix-free Krylov method.
  - Repeat until desired convergence is reached, and move to the next physical time step.
- Current solver uses a damped Newton line-search for the non-linear systems coupled with a GMRES Krylov method for the linear systems.

1D test case

1D Poisson test case to illustrate accuracy vs computational cost:

$$\frac{d^2u}{dx^2} = -20 + a\phi'' \cos\phi - a\phi'^2 \sin\phi \qquad (4)$$
$$a = 0.5,\ \phi(x) = 20\pi x^3$$



1D test case

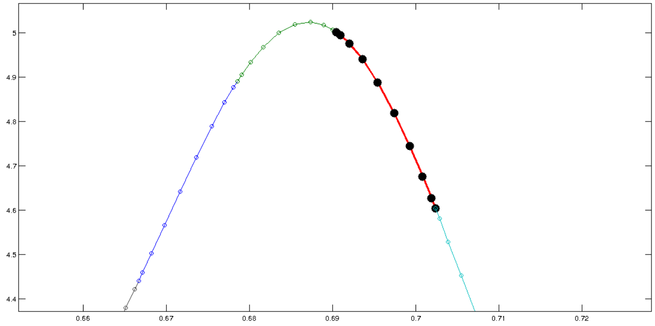


Figure : Close up of a single element with a 9th order polynomial basis.

1D test case

- For an ideal numerical method, computational cost is linearly proportional to the number of unknowns (degrees of freedom).
  - *e.g. 10 cells with 10 quadrature nodes compared to 50 cells with 2 quadrature nodes.*
- The end result is achieving equivalent accuracy with less computational expense or higher accuracy at similar computational expense compared to traditional finite volume methods.

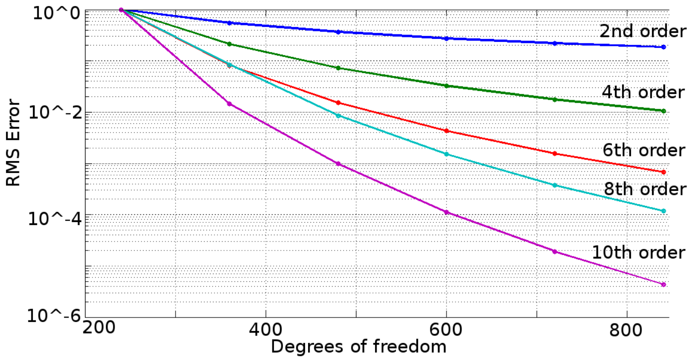
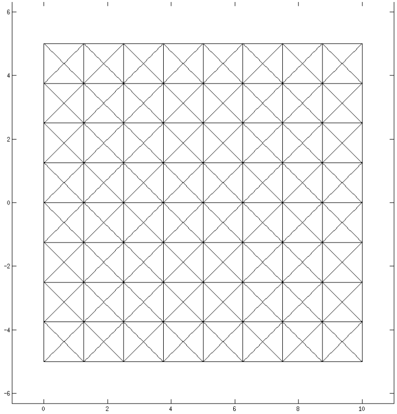
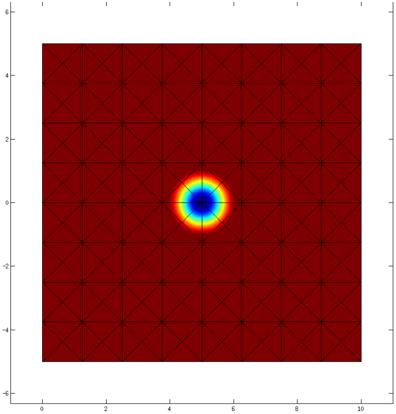


Figure : Error for varying order of accuracy with constant DOFs on 1D test case.

Test case - Isentropic vortex



(a) Coarse mesh for vortex case.



(b) Initial vorticity.

Test case - Isentropic vortex

- Non-dissipative vorticity convection is essential for these simulations.
- Test case of Yee et al (1999) for a convecting vortex is an exact solution for the compressible Euler equations. Free-stream conditions are

$$\rho = 1, u = u_\infty, v = v_\infty, p = 1$$

with an initial perturbation

$$(du, dv) = \frac{\beta}{2\pi} \exp\left(\frac{1-r^2}{2}\right)[-(y-y_0),(x-x_0)]$$

$$T = 1 - \frac{(\gamma-1)\beta^2}{8\gamma\pi^2}\exp(1-r^2)$$

$$\rho = T^{\frac{1}{\gamma-1}}$$

$$p = \rho^\gamma$$

for vortex center (x_0, y_0), and distance from center $r = \sqrt{(x-x_0)^2 + (y-y_0)^2}$.

Test case - Isentropic vortex - 1st order (c.f. 2nd order FV)

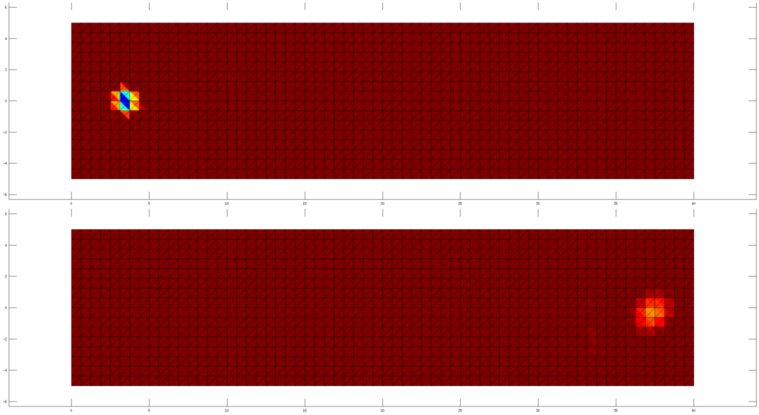


Figure : Vortex transport over 35 characteristic lengths, $O(\Delta x)$.

Test case - Isentropic vortex - 2nd order

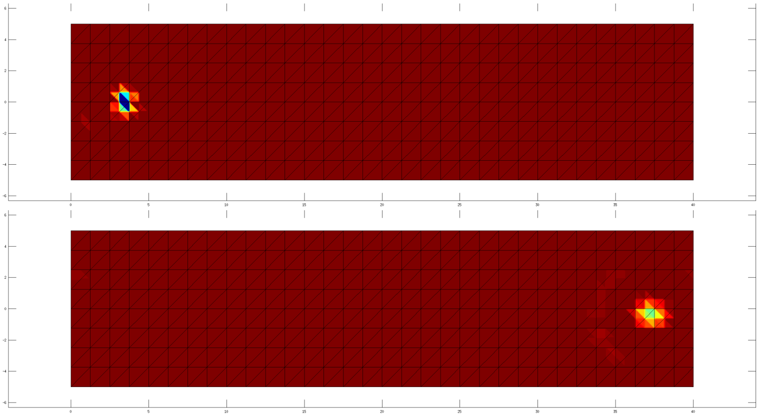


Figure : Vortex transport over 35 characteristic lengths, $O(\Delta x^2)$.

Test case - Isentropic vortex - 3rd order

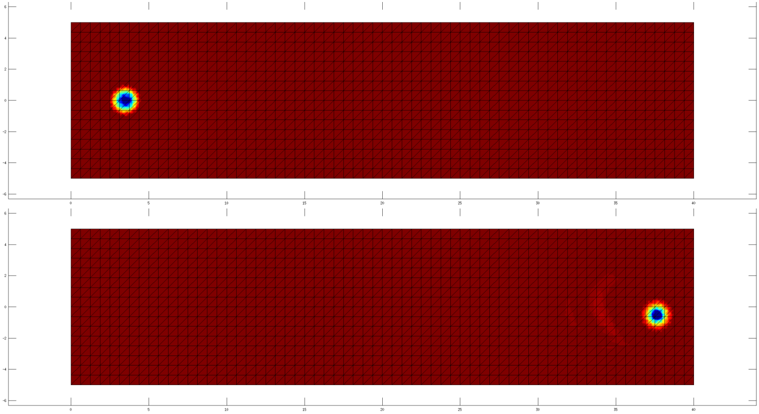


Figure : Vortex transport over 35 characteristic lengths, $O(\Delta x^3)$.

Test case - Isentropic vortex - 4th order

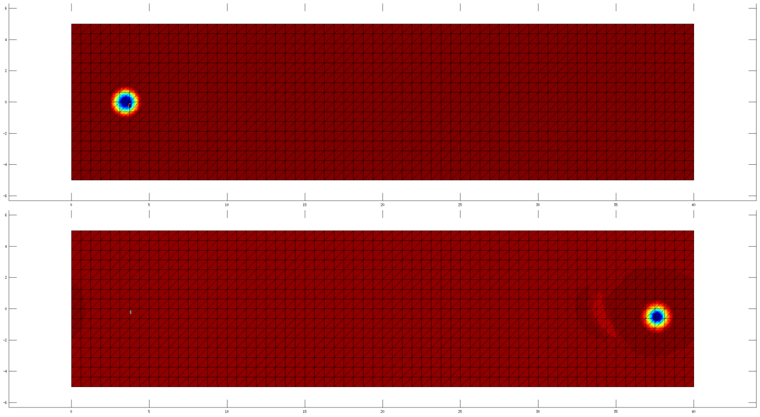


Figure : Vortex transport over 35 characteristic lengths, $O(\Delta x^{4})$.

Test case - Isentropic vortex order of accuracy

- L_2 norm of kinetic energy losses for isentropic vortex convection.

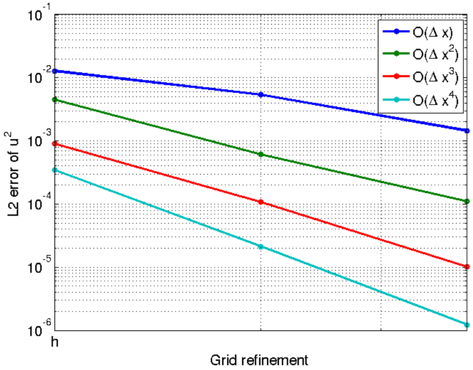
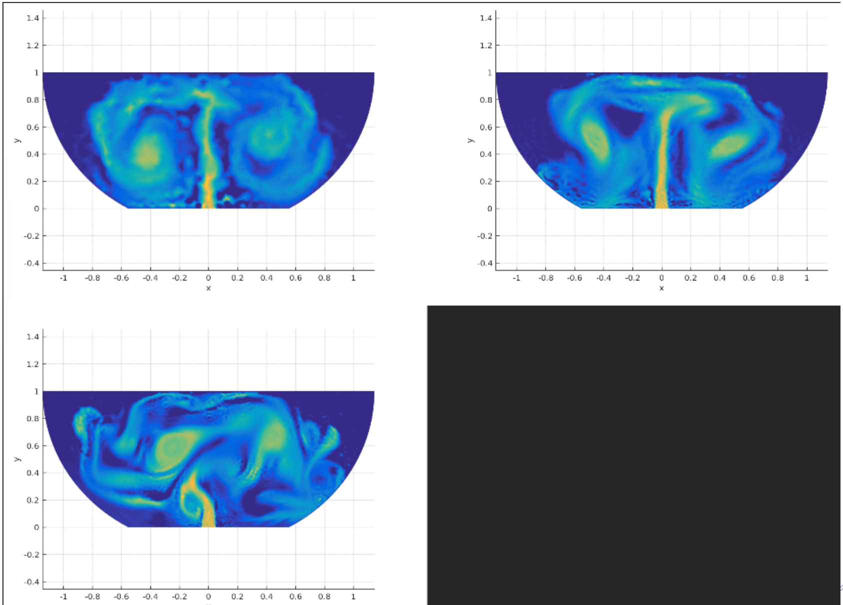


Figure : Solution accuracy versus grid refinement, for levels h, $h/2$, and $h/4$.

AIAA 2016 2D cargo hold results



AIAA 2016 2D cargo hold results

Uncertainy Quantification for Cargo Hold Fires, DeGennaro, Lohry, Martinelli, & Rowley, *57th AIAA Structures, Structural Dynamics, and Materials Conference*, San Diego CA, Jan. 2016.

- Two objectives of this study:
  - Assess the feasibility of using DG methods for buoyancy-driven flows,
  - Use uncertainty quantification techniques to analyze statistical variations in flows.

AIAA 2016 2D cargo hold results

- The mock fire sources were chosen to vary based on 2 parameters: fire strength and location.
  - Fire location was chosen to vary between the centerline and the far right wall, exploiting the symmetry of the geometry.
  - Fire strength was chosen to vary between a weak, slowly rising plume and a faster rising plume.
- 5×5 parameter sweep performed for these 2 parameters.
- Simulations performed with 3rd order elements (10 nodes per 2D cell) with approximately 1,500 triangular cells, or 15,000 nodes. All boundary conditions are isothermal non-slip walls. Time integration by 3rd order backward difference formula (BDF).

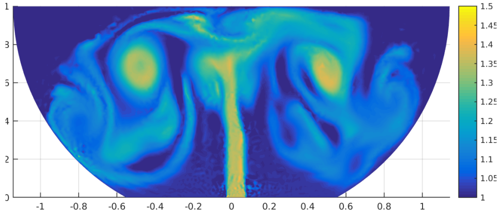
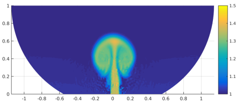
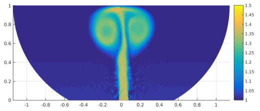


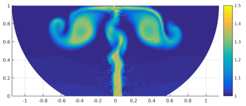
Figure : Flow driven by a heat source in a 2D cross-section. Colormap shown is temperature normalized by the initial bulk temperature.

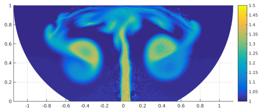
AIAA 2016 2D cargo hold results

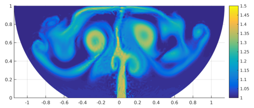
- Time evolution of temperature field:


(a) $t = 2s$.


(b) $t = 3s$.


(c) $t = 4s$.


(d) $t = 5s$.


(e) $t = 6s$.

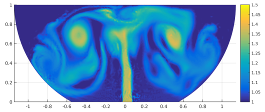
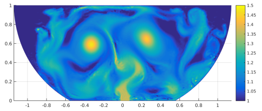
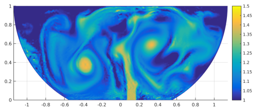

(f) $t = 7s$.

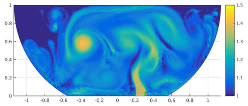
Figure : Temperature field time evolution for $T_s = 1.486$, $x_s = 0.024$ case.

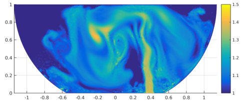
AIAA 2016 2D cargo hold results

- Variation of fire source location:


(a) $x_s = 0.024m$.


(b) $x_s = 0.116m$.


(c) $x_s = 0.262m$.


(d) $x_s = 0.387m$.

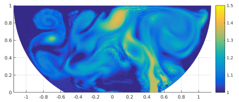
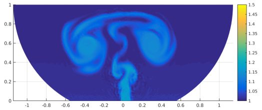
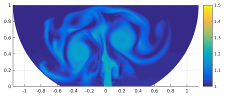

(e) $x_s = 0.480m$.

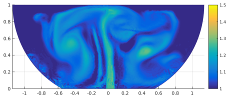
Figure : Temperature fields for $T_s = 1.486$ source at the 5 source locations, time $t = 10s$ after startup.

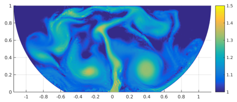
AIAA 2016 2D cargo hold results

- Variation of fire source temperature:


(a) $T_s = 1.214$.


(b) $T_s = 1.269$.


(c) $T_s = 1.350$.


(d) $T_s = 1.431$.

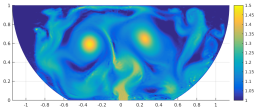

(e) $T_s = 1.486$.

Figure : Temperature fields at $x_s = 0.024m$ for the 5 values of temperature source, time $t = 10s$ after startup.

AIAA 2016 2D cargo hold results

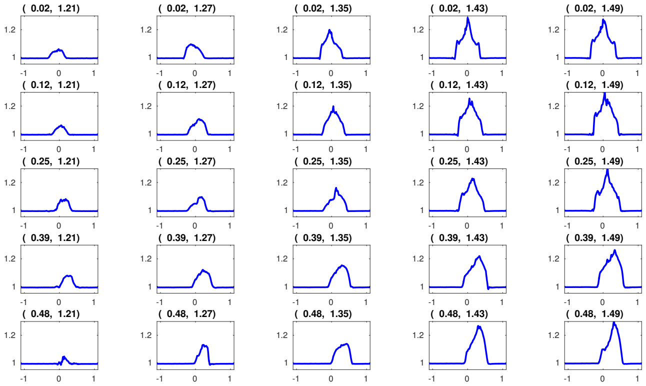


Figure : Time-averaged ceiling temperature distributions collected at the 25 quadrature nodes. Each subtitle corresponds to the parameter pair (x_s, T_s).

3D isentropic vortex

- Current work is on verification and validation of the full 3D problem.

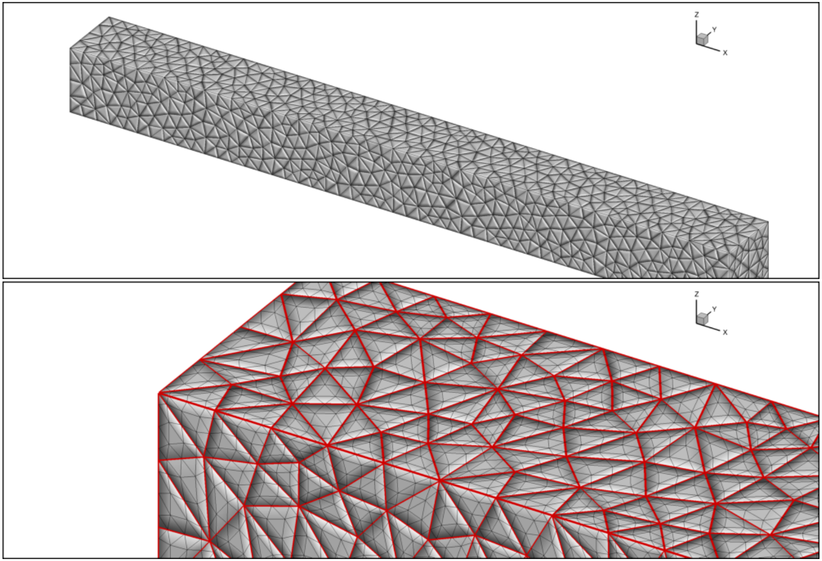
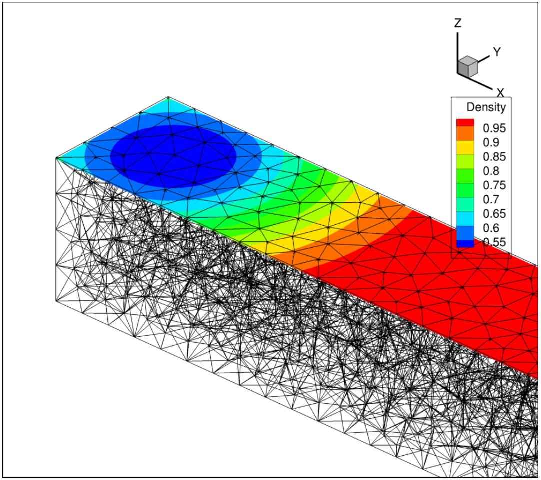
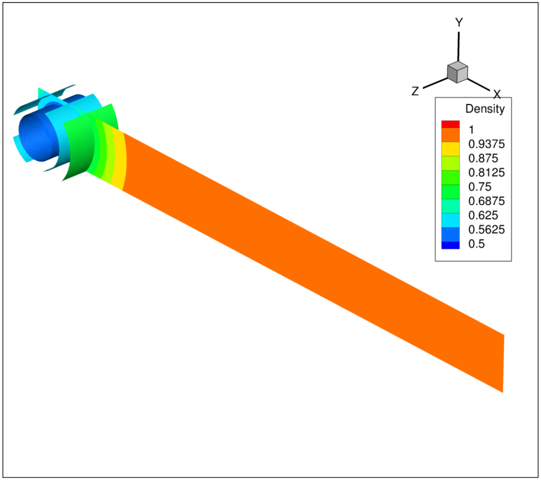


Figure : Comparison of mesh and 4th order elements.

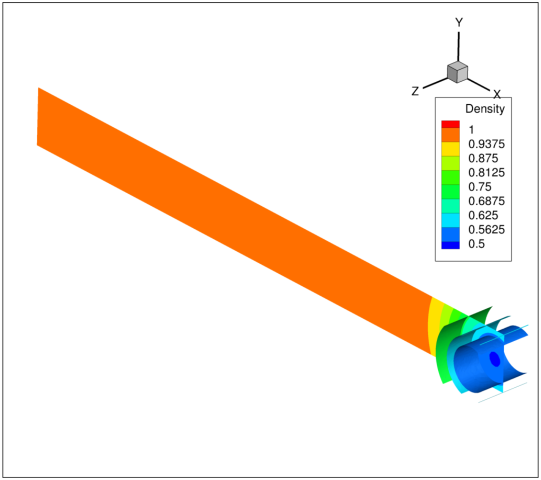
3D isentropic vortex



3D isentropic vortex



3D isentropic vortex



3D driven cavity

- Standard test case for viscous CFD. The "lid" of the cavity drives circulation through viscous entrainment similar to the buoyancy-driven instabilities.

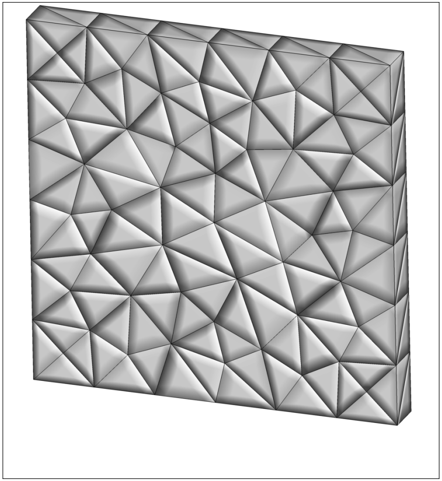


Figure : 354 cells 3D, 6x6x1 mesh.

3D driven cavity

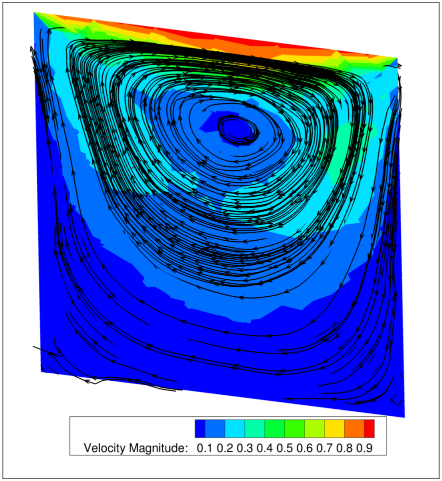


Figure : 1st order, 354 cells.

3D driven cavity

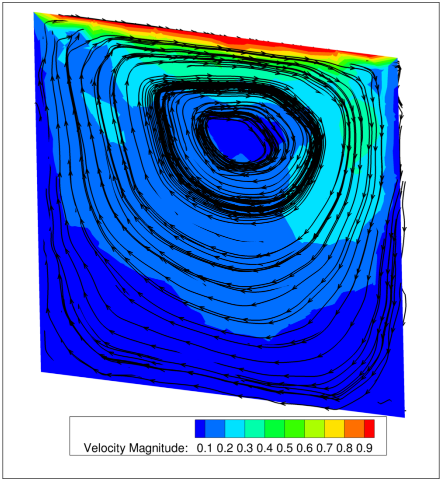


Figure : 2nd order, 354 cells.

3D driven cavity

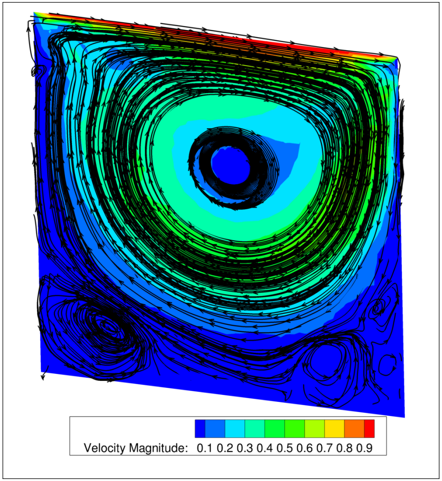


Figure : 3rd order, 354 cells.

3D driven cavity

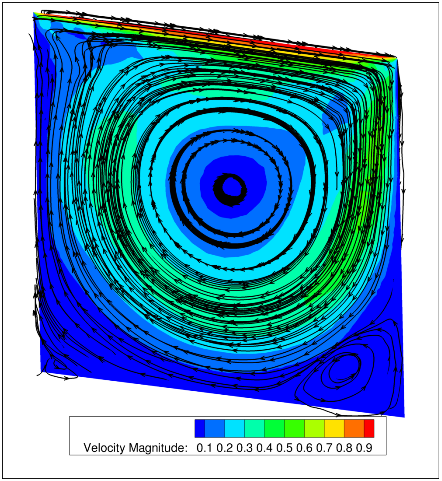
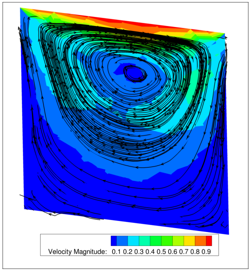
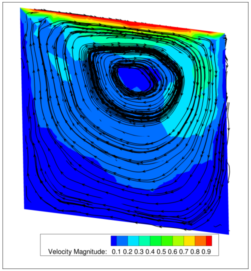
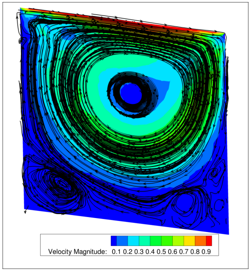


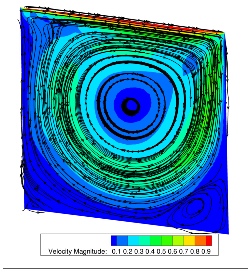
Figure : 4th order, 354 cells.

3D driven cavity









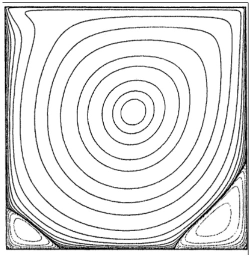
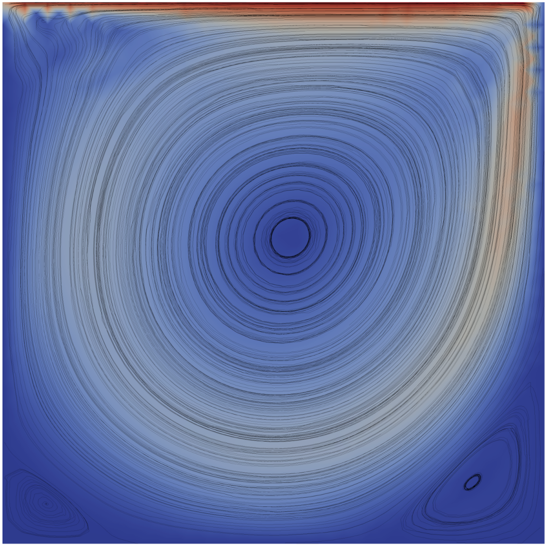
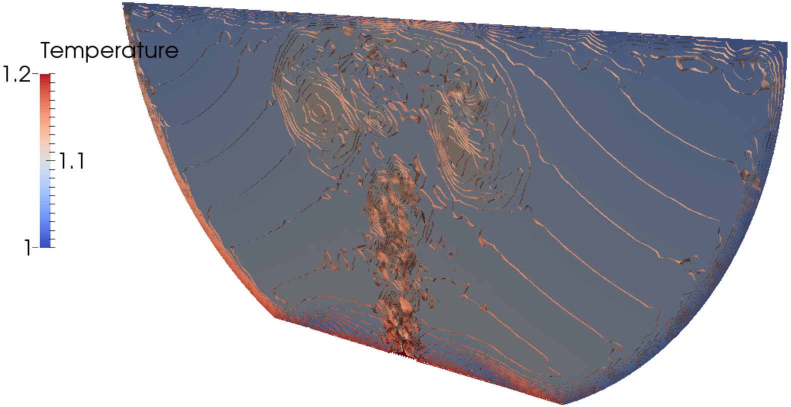


Figure : 3D DG solution with 354 cells c.f. Bruneau & Saad (2006), 1024 × 1024 grid.

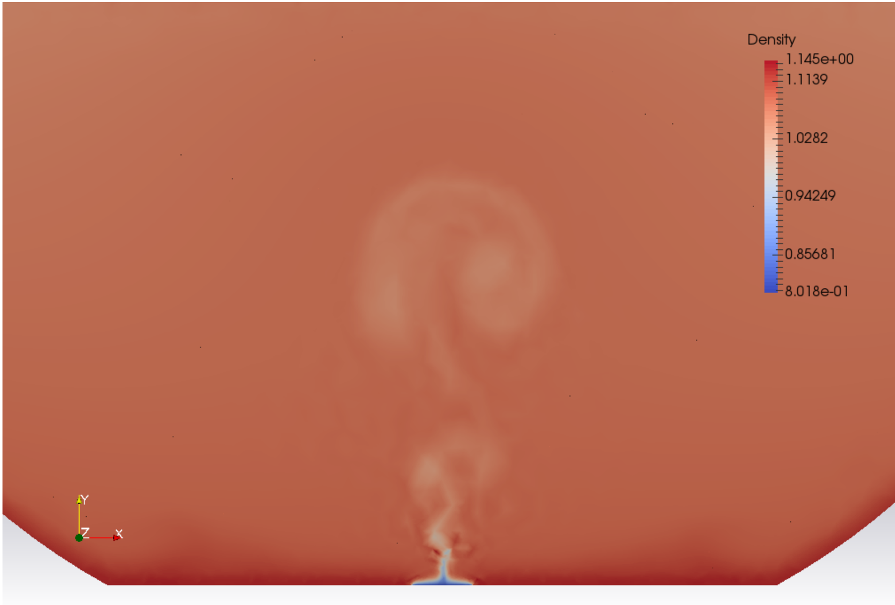
3D driven cavity



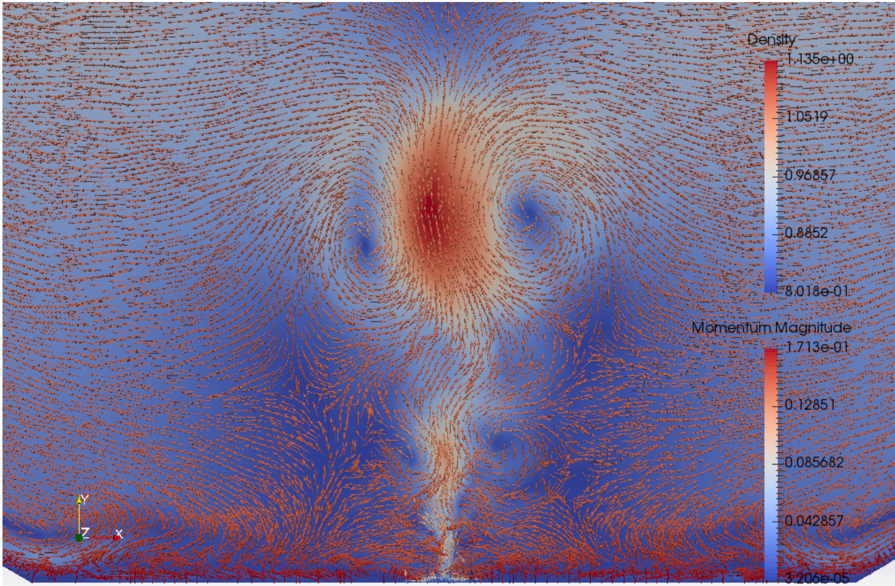
3D B707 cargo hold



3D B707 cargo hold



3D B707 cargo hold



3D B707 cargo hold

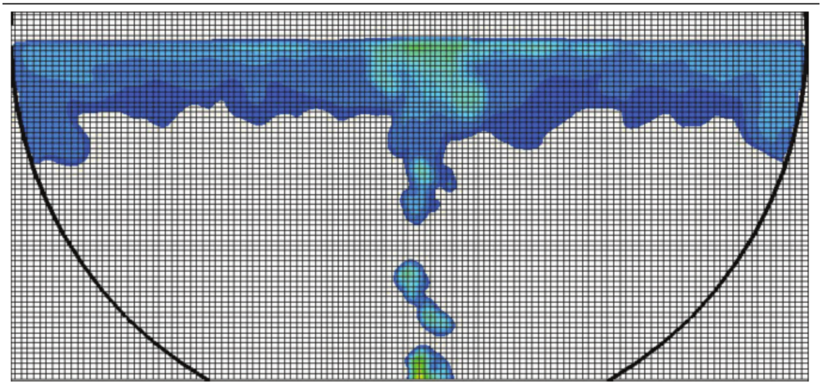


Figure : FDS, *Oztekin*

Software design aspects of a Discontinuous Galerkin solver

Aspects of the Discontinuous Galerkin solver:

- **Core flow solver** (*it works*):
    - 3D spatial discretization with unstructured meshes and arbitrary order of accuracy, cubature and quadrature rules for evaluating DG terms in the RHS, time integration, . . .
- **Bells and whistles** (*it's useful for complex problems*):
    - Interface to PETSc libraries to handle distributed memory parallelism, nonlinear and linear algebra, Jacobian-Free Newton-Krylov methods.
    - Error-adaptive implicit time stepping.
    - User implemented boundary conditions and volumetric sources to model fires with easy hooks via boost::dll.
    - Separation of discretization details from flow equations: very simple to switch between 1D, 2D, and 3D.
    - Implementation of traditional 2nd order finite volume method.
    - Wide variety of LES and RANS models.

Design

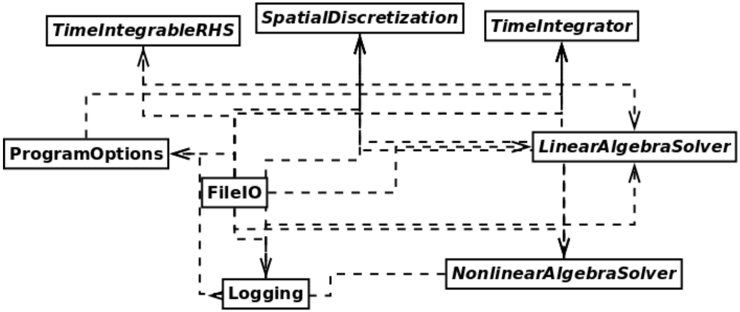
Major software components for DG solver:

- Evaluation of right-hand-side spatial discretization for Discontinuous Galerkin method.
- Time integration
- Nonlinear algebra solver
- Linear algebra solver
- Parallel communication
- File i/o
- User input options
- User-defined boundary/volume functions
- Logging of residuals, debugging info.

Lots of moving parts.

Design patterns - mediator

- Making everything work together without becoming a tightly coupled mess is hard.

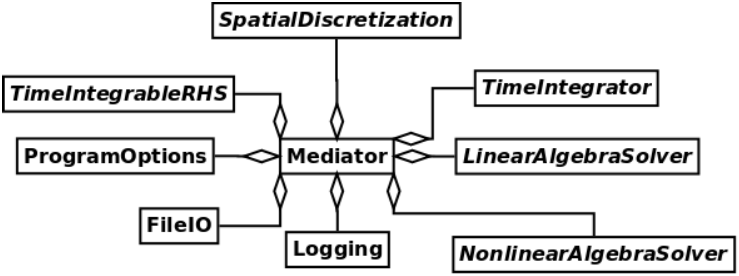


Spaghetti.

- $O(N)$ mutually interacting components require $O(N^2)$ communication complexity.

Design patterns - mediator

- The **mediator** [1] design pattern encapsulates interactions between classes, which reduces coupling by requiring all communication go through one class.
- Much easier to extend functionality and refactor existing code.
  - *Many-to-many* relationship becomes *one-to-many*.



Un-spaghetti'd mediator design pattern.

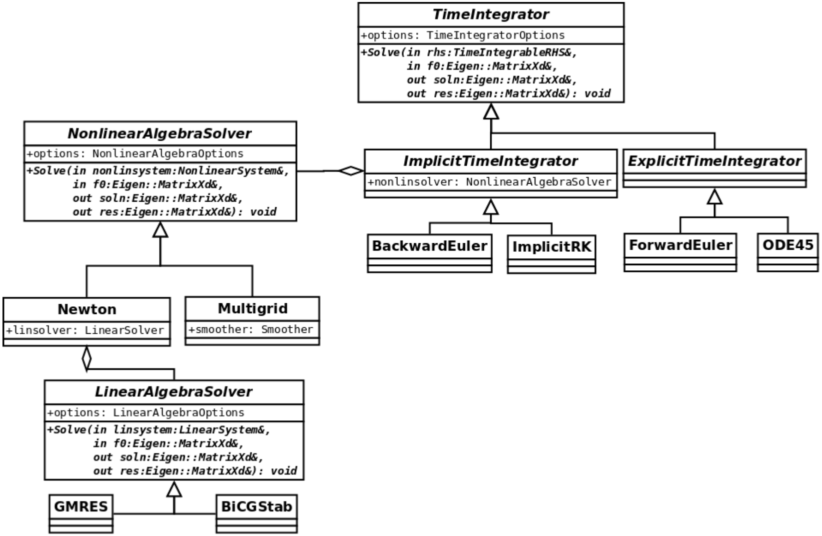
- I prefer the term **puppeteer** from Rouson et al.[2]

[1] Design Patterns: Elements of Reusable Object-Oriented Software, 1994

[2] Scientific Software Design: The Object-Oriented Way, 2011

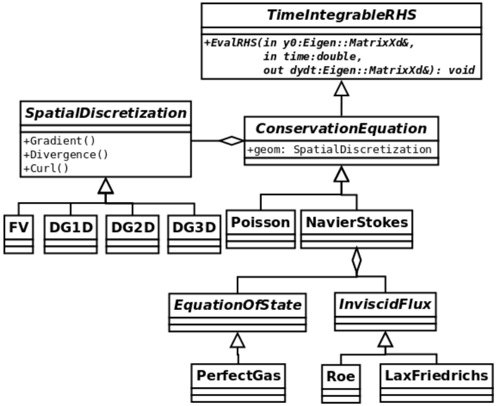
Design - time stepping and algebraic solution hierarchy

- Inheritance or composition where they make sense.
- Time integrators/solvers totally decoupled from spatial residual evaluations:


TimeIntegrator
+options: TimeIntegratorOptions
+Solve(in rhs:TimeIntegrableRHS&, in f0:Eigen::MatrixXd&, out soln:Eigen::MatrixXd&, out res:Eigen::MatrixXd&): void
NonlinearAlgebraSolver
+options: NonlinearAlgebraOptions
+Solve(in nonlinsystem:NonlinearSystem&, in f0:Eigen::MatrixXd&, out soln:Eigen::MatrixXd&, out res:Eigen::MatrixXd&): void
ImplicitTimeIntegrator
+nonlinsolver: NonlinearAlgebraSolver
ExplicitTimeIntegrator
BackwardEuler
ImplicitRK
ForwardEuler
ODE45
Newton
+linsolver: LinearSolver
Multigrid
+smoother: Smoother
LinearAlgebraSolver
+options: LinearAlgebraOptions
+Solve(in linsystem:LinearSystem&, in f0:Eigen::MatrixXd&, out soln:Eigen::MatrixXd&, out res:Eigen::MatrixXd&): void
GMRES
BiCGStab

Design - right hand side evaluation

- Inheritance or composition where they make sense.
- Time integrators/solvers totally decoupled from spatial residual evaluations:



Design patterns - registry

- **Mediator** becomes responsible for creation and ownership of various objects.
- I frequently use a singleton **registry** to store factory methods for abstract classes with multiple implementations:

```
template<class ClassType>
class Registry {
public:
  using Factory = ClassType* (*)(); // C++11 variant of typedef
  static Registry<ClassType>& Get() {
    static Registry<ClassType> instance;
    return instance; }
void Register (const std::string &name, Factory factory) {
    registry.insert(std::make_pair(name, factory)); }
...
private:
  Registry() {}
  ~Registry() {}
  std::map<std::string, Factory> registry;
```

```
 Registry<SomeClasstype>::Get().lookup("MyClassName") ...
```

Design patterns - registry

- Concrete classes can then register factories for themselves:

```
class BoundaryCondition {
 public:
  virtual void ApplyBC(...) = 0; ...

struct RegisterBC { // Functor for factory registration
  RegisterBC(string type,Registry<BoundaryCondition>::Factory factory)
    {Registry<BoundaryCondition>::Get().Register (type, factory);}
...
template<typename T >// A factory function for BCs
BoundaryCondition *BoundaryConditionFactory() {
    return new T; }
```

```
class BCWallViscousIsothermal : public BoundaryCondition{...}
static RegisterBC isowallinstance("BCWallViscousIsothermal",
           BoundaryConditionFactory<BCWallViscousIsothermal>);
```

```
// Returns a pointer to BC object of given name.
auto Registry<BoundaryCondition>::Get()
      .lookup("BCWallViscousIsothermal");
```

- Extension of these factory methods also enables potentially complex initialization on creation (RAII).

Unit testing

- I'm a big proponent of as much automated testing and test-driven development as possible.
- Test complexity in CFD codes ranges from very small scale unit tests to full scale engineering simulation tests.
- **Small-scale unit test:** Function for the 1D Legendre polynomial used frequently for interpolation and numerical integration:

```
Eigen::VectorXd LegendrePolynomial(const Eigen::VectorXd& x,
                                   const unsigned int N);
```

- Unit test, here checking accuracy of the $4th$ order polynomial to the analytical equivalent:

```
TEST(PolynomialInterpolation,Legendre){
 double tol = 1e-10;
 unsigned int npoints = 20;
 unsigned int Norder = 4;
 Eigen::VectorXd xn;
 xn.setLinSpaced(npoints,-1,1);
 auto lpoly = LegendrePolynomial(xn,Norder);
 for (unsigned int i=0; i!=npoints; ++i){
   EXPECT_NEAR(
    1/8*(35*pow(xn(i),4.0)-30*pow(xn(i),2.0)+3.0),lpoly(i),tol);
 ...
```

Unit testing

- Test for an RK4 time integration scheme:

```
class RungeKutta4 : public TimeIntegrator {
  void Solve( TimeIntegrableRHS* rhs,
              const Matrix& y0,
              Matrix& soln );
```

- Testing a time integrator on a full CFD problem isn't necessary.
- Knowing that RK4 should be able to exactly integrate a 4th order ODE, set up a mock problem $y' = t^4$ from 0 to 10:

```
class RK4TestProblem : public TimeIntegrableRHS {
 void EvalRHS(const Matrix& y,const double t,Matrix& soln){
  soln=pow(t,4.0); }
...
TEST_F(TimeIntegration,RK4){ ...
 y0 = 0; t0 = 0; tfinal = 10;
 RK4Integrator -> Solve( RK4Test, t0, tfinal, y0, soln);
 EXPECT_NEAR( pow(t,5.0)/5.0, soln, tolerance );
```

Unit testing

- More complex integration example covering more code, testing a nonlinear algebra solver, which utlizes a Newton solver combined with a Krylov linear algebra solver.
- Solve nonlinear system $f(x,y) = [x^2 + y^2 - 10; 2 * x + y - 1]$ which has two solutions, $(-1, 3)$ and $(9/5, -13/5)$:

```
class NonlinearSystemTest : public NonlinearSystem {
  void EvalRHS(const Matrix& y,Matrix& soln){
   soln(0) = x*x + y*y - 10;
   soln(1) = 2*x + y - 1;
...
TEST(NonlinearAlgebra,NewtonGMRES){
 ...
 nonlinearsolver->Solve(nonlinearsystem, initial_guess,...)
 EXPECT_NEAR( -1, soln(0), tol );
 EXPECT_NEAR(  3, soln(1), tol );
...
 /* another test with different initial guess that
    should recover other solution. */
```

Unit testing

- **High level integration test example:** Flow simulation of a cylinder at a given Reynolds number. Quantities to test:
    - Zero mean lift.
    - Drag within experimental tolerances.
    - Strouhal number within tolerance.
- **Method of manufactured solutions:** Write out an exact solution for NS, derive the corresponding source terms to force that solution.
- Very high level integration (or verification and validation) tests are always necesesary to show we can get the right answer on test problems.
    - Also provide broad regression testing.
- They're of limited use for a developer: *if the test simulation fails, where in the code is the culprit?*
- **Low level tests:** Tells you precisely where in some procedural code you have an error.
- **Medium level tests:** Tests assumptions about the integration between various moving parts.
- **High level tests:** Make sure real engineering problems are correct.
- **Prefer writing automated tests over debugging.**

Unit testing

- **Subtle benefit to lots of low level testing:** Writing code that *can* be tested results in cleaner interfaces.(*Separation of concerns* or *single responsiblity principle* notions in software design.)

Very bad design of an RK method from a big production F90 code:

```
       subroutine executeRkStage
!      * executeRkStage executes one runge kutta stage.
       use blockPointers
       use constants
       use flowVarRefState
       use inputIteration
       use inputPhysics
       use inputTimeSpectral
       use inputUnsteady
       use inputDiscretization
       use iteration
       use inputPhysics
 ...
```

- Source file is *578 lines long*, subroutine *takes no arguments*, relying entirely on 10 effectively global module, using 40+ of those global state variables.
- *Totally untestable, unreadable, unmaintainable, tightly coupled, error-prone, unmodifiable, overly complicated code.*

Unit testing

- Compared to a design of similar functionality from my DG code, but springing from test-driven development:

```
// 5 stage 4th order low storage explicit RK scheme.
class LowStorageRK54 : public TimeIntegrator {
public:
    void Solve(TimeIntegrableRHS* rhs,       // object for evaluating RHS
               const MatrixXd&    f0,        // f(t=0) initial condition
               MatrixXd&          soln,      // f(t=tfinal) solution
               MatrixXd&          residual)  // residual of last iteration
```

- Totally self-contained time stepping.
- No coupling to the physical problem or to the spatial discretization.
- Works the same on a single scalar ODE for a unit test as it does on a 10^9 unknown CFD simulation running on 1,000 cores.

Ongoing solver development

2D work completed:

- Established that high-order-accurate discontinuous Galerkin methods can be used for simulating buoyancy-driven flows such as those seen in cargo hold fires, using unstructured meshes suitable for arbitrary geometries.
- Demonstrated the use of these simulations in an uncertainty quantification framework to aid in fire sensor placement.

Current work is on extending this to a 3D solver for full cargo hold simulation:

- Functioning:
  - 3D unstructured flow solver, spatial discretization with arbitrary order of accuracy.
  - Parallel scaling.
  - Jacobian-Free Newton-Krylov for solution of non-linear algebra.
  - Implicit time integration for high order temporal accuracy and large time step stability.
  - 3D viscous effects
- Work in progress:
  - Full testing of 3D buoyancy-driven effects.
  - Implementation of Large Eddy Simulation (LES) models.
  - Full cargo hold simulations for validation.
  - Direct quantitative comparisons between OpenFOAM/FDS and this DG work for validation.